

Introduction

Previously on A-Series systems, the maximum number of active processes contending for the processor was 4095. On NX5820 and NX6820 systems, this limit was increased to 32767. This increase brought to light an algorithmic performance problem related to the operating system CONTROLLER process monitoring the state of so many processes; this information is displayed at the operator display terminals (ODTs). Affected operator commands used to display process status include J (job structure view of processes), A (active processes), W (processes waiting for operator action), S (processes scheduled for execution), DBS (database processes), and LIBS (library processes).

Previously when the total number of concurrently active processes was low (less than 500), the CONTROLLER process required less than 6% of the processor power of a single processor to display process state at the ODT. When the total number of concurrently active processes was increased to 8000, the CONTROLLER process consumed an entire processor just to display process state. When the total number of concurrently active processes was increased to 32000, the CONTROLLER process consumed an entire processor to display process state and could not keep up with the requested display refresh rates. When the CONTROLLER process was asked to sort active processes by CPU rate, the CONTROLLER process could only display process state information in intervals of several minutes.

System processes on A-Series systems are organized in a parent-child hierarchy. When a user establishes a session or starts a job, this process spawns off children processes. The children in turn may spawn further processes. The entire set of processes associated with a session or a job is known as a process family. For example, the following is output for the J operator command for a single process family:

```
--Mix-Pri----- JOB ENTRIES -----
59641  50  Lib *SYSTEM/JAVASERVLETLIB ON JAVATEST
59646  50      ..SERVLET/API/PROCESS/NEW/REQUESTS
59645  50      ..*OBJECT/JAVA ON JAVATEST
59661  50      ....P59645/9/1/"TimerThread"
59660  50      ....P59645/8/1/"Worker2"
59659  50      ....P59645/7/1/AWAITING REUSE
59657  50      ....P59645/6/1/"Thread-0"
59655  50      ....P59645/5/1/"SessionMgrThread"
59651  50      ....P59645/4/1/"Finalizer"
59650  50      ....P59645/3/1/"Reference Handler"
59649  50      ....P59645/2/1/"Signal dispatcher"
```

In this example, the library job process *SYSTEM/JAVASERVLETLIB ON JAVATEST spawned 2 child processes – SERVLET/API/PROCESS/NEW/REQUESTS and *OBJECT/JAVA ON JAVATEST. The first child process had no offspring while the second child process spawned 8 child processes (grandchildren of the library job process *SYSTEM/JAVASERVLETLIB ON JAVATEST).

The data structure that the system uses to organize process families in a parent-child structure is a general tree (also known as an abstract data type tree). A general tree is constructed of a parent node that is linked to 0 or more children nodes. Each of the children nodes are linked to 0 or more children nodes of their own. General trees are ideally suited for linking hierarchal entities like process families. A comprehensive description is included later in this form.

On A-Series systems, the CONTROLLER process is responsible for periodic update and display of process state system-wide. Each periodic update displays status for up to, for example, 127 processes. The next periodic update picks up where the previous update left off. The A-Series CONTROLLER

process did not scale for a very large number of active processes because each display of updated process status information required a status picture of the entire process family tree (a general tree) to be built. For a system that has only, for example, 150 active processes, the effort to build the status picture of the process family tree is rather insignificant when displaying status information for only 127 of these processes. However for a system that has 32000 active processes, the effort to build the status picture of the entire process family tree is very significant when only 127 of these items will be displayed.

Solution Description

This invention provides a mechanism where preorder traversal of a general tree can start and stop at any point in that tree. Thus when the A-Series CONTROLLER process displays status information for 127 processes, a status picture of a partial family tree consisting of 127 processes is all that is required. At the next periodic update of process state, this invention provides a mechanism where the status picture can pick up just after the last process that was displayed in the previous status display update.

Key points regarding this invention include:

1. In general, this mechanism is most suited to an environment where the process that requires preorder tree traversal is not the process that "owns" the tree data structure. For example, the CONTROLLER process is displaying process status information and the A-Series MCP KERNEL contains the process tree structure.
2. The tree is dynamic. Tree nodes may have been created and/or destroyed in between calls to retrieve the tree structure.
3. The tree data structure may be very large. Without this mechanism, a process that is consuming tree nodes may require frequent updates of the entire tree structure so that the node information does not go out of date. This requires a substantial amount of data to be retrieved and transferred while only a small amount of data is actually consumed.
4. The individual node values need not exhibit and need not be inserted in any apparent order within a single group of sibling nodes.

The general tree traversal continuation mechanism consists of 3 items:

1. A node structure that includes unique node counter. For example, a timestamp could provide this functionality providing that the timestamp is of sufficient granularity to make each timestamp unique. This counter will be used for finding the correct starting point when the specified continuation node no longer exists.
2. A node structure that includes a backward node pointer to the parent node. While this backward pointer is not required as part of normal traversal of the tree, most implementations of trees already include this pointer because it has so many uses that are related to other tree operations.
3. A representation of the entire branch (known as a lineage) for the continuation node is required. This consists of the node, its parent, its parent's parent, and so on all the way to the root parent.

Using this invention, the following table illustrates the reduced CPU utilization realized by the CONTROLLER process on an NX5820 A-Series system:

CONTROLLER Update of Active Process Status Every 5 Seconds – ADM (ACTIVE 21) DELAY 5		
Number of Active Processes	CPU Utilization Prior to the Invention	CPU Utilization After the Invention
500	6%	1%
8000	100%	1%
32000	Could not do	1%

Features Recitation

1. External traversal of a general tree can stop and start at any branch in that tree. The external process needs only to retrieve the specific tree nodes that it will process.
2. If the continuation tree node no longer exists, this mechanism provides way to find what would be the next node process in the tree.
3. On a static general tree (nodes are neither created nor destroyed), standard traversal of this tree yields exactly the same structure as this mechanism would produce in a step-wise manner.
4. The continuation mechanism requires minimal data to indicate continuation information. This information is on the order of the depth of the tree (the number of levels from the root node to the continuation node).
5. Continuable partial preorder traversal of a general tree can be achieved step-wise (single node) or by returning multiple nodes.

Subject Matter Description and Drawings

Prior to discussion about the invention, a few definitions and concepts are in order.

Node – an element of a tree.

Root Node – a single node at level 0.

Child Node – a node at level 1 or greater descending from a parent node.

Parent Node – a node associated with a lower level node. Every child node has exactly 1 parent.

Sibling Nodes – nodes that are at the same level and share the same parent node.

Leaf Node – a node with 0 children.

Empty Tree – a tree consisting of 0 nodes.

General Tree – a set of nodes consisting of an empty tree or a root node and 0 or more subsets each of which is a tree.

Abstract Data Type Tree – general tree.

Forest – a set of 0 or more trees.

Binary Tree – a special type of general tree that consists of an empty tree or a root node and exactly 2 subsets each of which is a tree.

N-ary Tree – a special type of general tree that consists of an empty tree or a root node and exactly N subsets each of which is a tree.

Lineage – the path of nodes current node up to the root node (i.e., the node, its parent, its parent's parent, etc., up to the root node).

Level – the level of the root node is 0; the level of any other node is 1 + the level of its parent (i.e., the level is the number of nodes in the lineage – 1).

Traverse – an orderly way to visit each node in the tree exactly once in order to perform some operation on that node.

Preorder Traversal of a general tree – starting at the root node

-
1. visit node
 2. traverse each of the children subtrees

Postorder Traversal of a general tree – starting at the root node

1. traverse each of the children subtrees
2. visit node

Preorder Traversal of a binary tree – starting at the root node

1. visit node
2. traverse left subtree
3. traverse right subtree

Inorder Traversal of a binary tree – starting at the root node

1. traverse left subtree
2. visit node
3. traverse right subtree

Postorder Traversal of a binary tree – starting at the root node

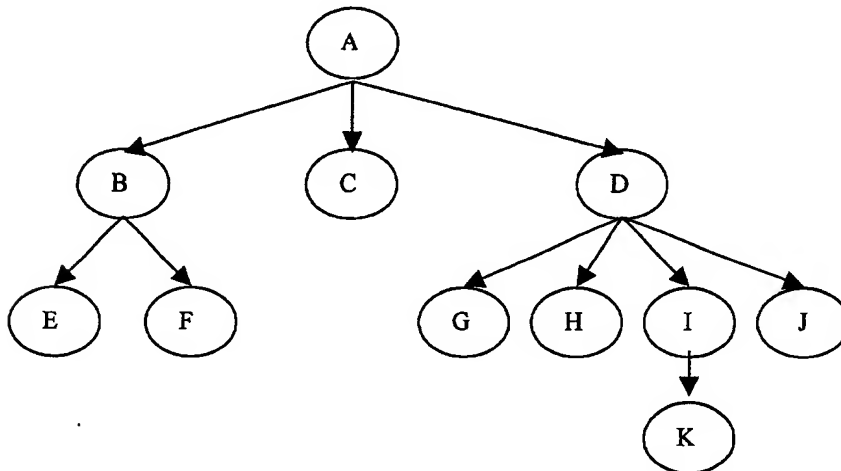
1. traverse left subtree
2. traverse right subtree
3. visit node

A tree is a fundamentally hierarchical structure. As such, a tree may be used to represent any model that exhibits hierarchy. These include:

1. Process family structure
2. Disk file directory structure
3. Process priority scheduling queues
4. Genealogical trees including family relationships among individuals, tribes, languages, etc.
5. Classification systems including the Dewey decimal system, taxonomic classification of plants and animals, etc.
6. Program structure (main program, procedures, nested procedures, etc.)
7. Breakdown of a manufactured product or service

The following example is used to illustrate the structure of a general tree, traversal of a general tree, implementation of a general tree using a binary tree including the associated node structures, and traversal of the binary tree used to implement the general tree.

Example 1 General Tree:



Preorder Traversal: A B E F C D G H I K J
 Postorder Traversal: E F B C G H K I J D A

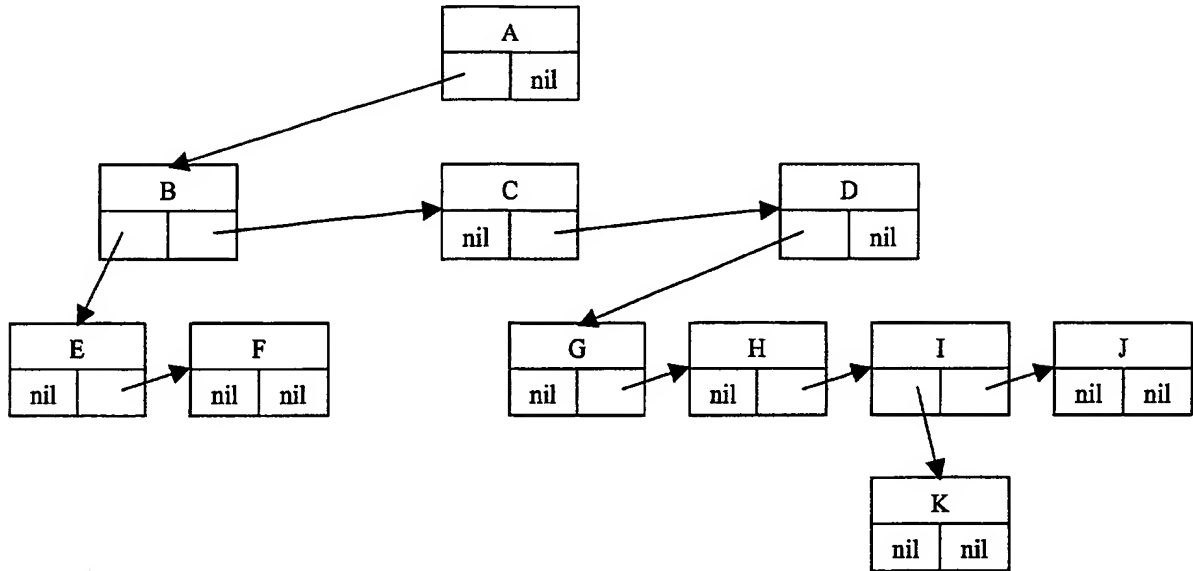
A general tree may be used to represent a directory structure where a file exists for each leaf node E F C G H K J. Preorder traversal of the tree in example 1 to retrieve all file names would yield:

A\B\E
 A\B\F
 A\C
 A\D\G
 A\D\H
 A\D\I\K
 A\D\J

A binary tree can be used to represent a general tree. A binary tree is easily represented in programming languages as a uniform structure consisting of a data value field, a pointer reference to the first child node (the next generation), and a pointer reference to the next sibling node (the current generation). The following illustrates general node structure:

Data Value	
Child Ptr	Sibling Ptr

The following binary tree would represent the general tree shown in example 1:



Preorder Traversal: A B E F C D G H I K J
Inorder Traversal: E F B C G H K I J D A
Postorder Traversal: F E K J I H G D C B A

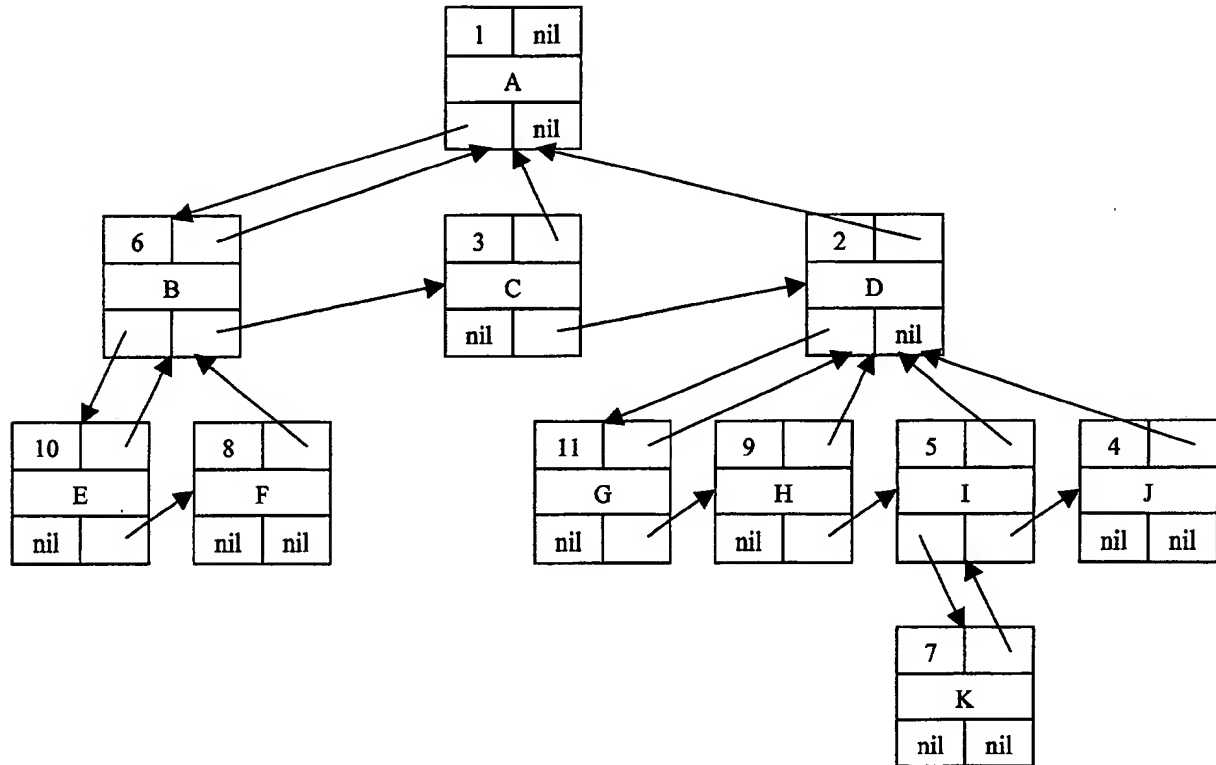
Note that when comparing tree traversal for the general tree shown in Example 1 versus the binary tree that represents that general tree, the following characteristics can be observed:

- Preorder traversal of a general tree is the same as preorder traversal of the binary tree that represents the general tree.
- Postorder traversal of a general tree is the same as inorder traversal of the binary tree that represents the general tree.
- There is no traversal of a general tree that corresponds to postorder traversal of the binary tree that represents the general tree.

To support a general tree preorder traversal continuation mechanism, the node structure must be expanded to store 2 additional items – a unique counter (incrementing or decrementing) and a parent pointer. The general structure of a node that supports continuation calls is:

Counter	Parent Ptr
Data Value	
Child Ptr	Sibling Ptr

The following binary tree with the new node structure would represent the general tree shown in example 1:



Note that for this general tree, the unique counter in the node structure does provide some semblance of order. In each case, this counter was incremented each time a node was inserted in the tree. From this structure using the non-decreasing node counter, one can see that the order of insertion of nodes was: A D C J I B K F H G E. Also note that for all children of a specific node (the child node and all sibling nodes at the same level), the counter value of each node decreases as one follows the sibling links.. This becomes important for external traversal of this tree as the linear ordering of sibling nodes allows for detection of where to pick up when the continuation node has been deleted.

For the process that owns the tree data structure (in the process's data environment), there are a number of fundamental operations and functions that can be performed on the tree. These include:

Create_Root_Node (root_ptr, node_value)

Insert_Node (parent_ptr, node_value)

Delete_Node (node_ptr)

Preorder_Traverse_Tree (root_ptr)

Postorder_Traverse_Tree (root_ptr)

Pseudocode for operations pertinent to this design for the associated node linkage and counter use include:

```

5      PROCEDURE Create_Root_Node (root_ptr, node_value)
      INTEGER root_ptr BY REFERENCE;
      INTEGER node_value BY VALUE;
      BEGIN
10         Allocate_Node (root_ptr, node);
            Increment (node_counter);
            node[root_ptr].Counter := node_counter;
            node[root_ptr].value := node_value;
            node[root_ptr].sibling_pointer := nil;
15         node[root_ptr].child_pointer := nil;
            node[root_ptr].parent_pointer := nil;
      END;

```

Note in procedure **Insert_Node** that for each time a node is inserted into the tree, the global node counter is incremented and the new node becomes the child node while the previously linked children node get pushed to the front of the sibling node chain. This implies some type of ordering. If the global node counter is incremented, the child node and all sibling nodes have an associated node counter value that is decreasing when following the sibling links; similarly if the global node counter is decreasing, the child node and all sibling nodes have an associated node counter value that is increasing.

```

20      PROCEDURE Insert_Node (parent_ptr, node_value);
      INTEGER parent_ptr, node_value BY VALUE;
      BEGIN
30         INTEGER node_ptr, ptr;
            Allocate_Node (node_ptr, node);
            Increment (node_counter);
            node[node_ptr].Counter := node_counter;
35         node[node_ptr].value := node_value;
            ptr := node[parent_ptr].child_pointer;
            node[parent_ptr].child_pointer := node_ptr;
            node[node_ptr].sibling_pointer := ptr;
            node[node_ptr].child_pointer := nil;
40         node[node_ptr].parent_pointer := parent_ptr;
      END;

```

Two versions of procedure **Delete_Node** are shown to illustrate differences on how to handle deleting a node that has offspring. The first version of procedure **Delete_Node** requires that the node being deleted has no offspring nodes. The second version of procedure **Delete_Node** below has a recursive implementation where a node and its entire offspring are deleted. To delete a node and all of its offspring nodes, a recursive post-order traversal of the offspring tree is shown.

```

50      PROCEDURE Delete_Node (node_ptr);
      INTEGER node_ptr BY VALUE;
      % This version of Delete_Node requires the node to be deleted has no offspring.
      BEGIN
55         IF node[node_ptr].child_pointer = nil THEN
            BEGIN
                node[node_ptr].parent_pointer.child_pointer :=
                    node[node_ptr].sibling_pointer;
                Deallocate_Node (node_ptr);
60         END ELSE
            RETURN ("error: cannot delete node while offspring exist")
      END;

```

```

PROCEDURE Delete_Node (node_ptr);
INTEGER node_ptr BY VALUE;
% This version of Delete_Node deletes the node and its entire offspring.
BEGIN
5   PROCEDURE Delete_Subtree (ptr);
      BEGIN
          IF node[ptr].child_pointer <> nil THEN
              Delete_Subtree (node[ptr].child_pointer);
          IF node[ptr].sibling_pointer <> nil THEN
10      Delete_Subtree (node[ptr].sibling_pointer);
          Deallocate_Node (ptr);
      END;

      IF node[node_ptr].child_pointer <> nil THEN
15      Delete_Subtree (node[node_ptr].child_pointer);
      node[node_ptr].parent_pointer.child_pointer :=
          node[node_ptr].sibling_pointer;
      Deallocate_Node (node_ptr);
20  END;

```

Traversal of general trees in the environment in which the tree resides is a very trivial operation. The following procedures **Preorder_Traverse_Tree** and **Postorder_Traverse_Tree** illustrate display of node values (visit node) using recursive versions of preorder and postorder traversal of a general tree. For these procedures, recursion can be eliminated using a recursion simulation stack. These standard traversal techniques are NOT the subject of this patent.

```

PROCEDURE Preorder_Traverse_Tree (node_ptr);
INTEGER node_ptr BY VALUE;
BEGIN
30  DISPLAY (node[node_ptr].node_value);
      IF node[node_ptr].child_pointer <> nil THEN
          Preorder_Traverse_Tree (node[node_ptr].child_pointer);
      IF node[node_ptr].sibling_pointer <> nil THEN
          Preorder_Traverse_Tree (node[node_ptr].sibling_pointer);
35  END;

```

The procedure **Postorder_Traverse_Tree** uses an inorder traversal of the binary tree that represents the general tree. This is equivalent to postorder traversal of a general tree.

```

PROCEDURE Postorder_Traverse_Tree (node_ptr);
INTEGER node_ptr BY VALUE;
% Postorder traversal of a general tree is equivalent to inorder traversal of the
% binary tree that represents that general tree.
45  BEGIN
      IF node[node_ptr].child_pointer <> nil THEN
          Postorder_Traverse_Tree (node[node_ptr].child_pointer);
      DISPLAY (node[node_ptr].node_value);
      IF node[node_ptr].sibling_pointer <> nil THEN
50      Postorder_Traverse_Tree (node[node_ptr].sibling_pointer);
      END;

```

The continuation mechanism for partial preorder traversal of dynamic general trees is generally defined for an external process retrieving a snapshot of a tree structure that is not in its data environment. As such, external processes must pass 2 data structures: a structure that resembles a subtree (e.g., an array of nodes) and a pointer to a specific node in that structure. For the specified continuation node in the subtree, the entire lineage of that node is also passed in the subtree using the parent pointer information in each node. Thus the amount of information required for continuation information is limited only to the level or generation of a specific node from the root node.

Returning subtrees for partial preorder traversal of nodes can be achieved using step-wise (single node) and multiple node operations. Single and multiple node partial traversal is made possible by the external node structure that contains the entire lineage for each node returned (i.e., the parent nodes were visited

before the current node); this corresponds to the preorder search visitation order that must take place internally to return such information.

This differs from partial postorder traversal of dynamic general trees in that the corresponding internal interface that returns postorder nodes visits parent nodes after visiting child nodes. Thus, partial postorder traversal of a general tree would be generally limited to step-wise (single node) operations if a tree-like structure is to be returned to the calling process. It is possible to return multiple nodes in a partial postorder traversal, but the associated algorithms are much more complicated space-wise (the complete lineage nodes that accompany each returned node are repeated in the passed subtree) or time-wise (multiple passes through the passed subtree must be performed to coalesce the various lineages into a unique set of nodes). In either case, the returned structure is not a tree. As such, partial postorder traversal of a dynamic general tree is NOT the subject of this patent.

Two external interfaces that support partial preorder traversal of dynamic general trees are defined:

Single_Step_Preorder_Traverse_Tree (ext_node_ptr, ext_nodes);

Partial_Preorder_Traverse_Tree (ext_node_ptr, ext_nodes);

The nodes returned in the **ext_nodes** array minimally require 3 fields when single-stepping through the traversal (node value, parent pointer, and node counter value). The child pointer and sibling pointer are not required to perform a preorder traversal because the tree nodes are returned sequentially using preorder. However some external applications may still require the complete node structure and linkage, so that is what will be shown in the pseudocode examples.

The parameters for these external interfaces are defined in a manner that is used as both input to the internal procedures supporting traversal and output results that contain the partial tree structure. For example, if these procedures are called with the **ext_node_ptr** parameter set to a nil value (as defined by the internal structure), the resulting **ext_nodes** parameter contains a partial tree that starts with the node at the **root_ptr** of the internal tree; the resulting **ext_node_ptr** points to the next node from which to continue preorder traversal. As an external process consumes nodes, it is responsible for updating the external node pointer value (**ext_node_ptr**) so that subsequent calls on this interface return the next set of nodes. When using the single step version of this traversal (**Single_Step_Preorder_Traverse_Tree**), update of **ext_node_ptr** is not necessary. The most important invariants that enables the output of the previous call to be used as input for the current call is that for each node that is returned in the partial tree, there exists a complete lineage of nodes all of the way back to the root node and a unique counter value for each node. The complete lineage is a characteristic of preorder tree traversal that is not found in postorder tree traversal, and that is why this patent is limited to partial preorder tree traversal.

Characteristics of the **ext_nodes** array and **ext_node_ptr** after calling the external traversal interfaces include:

- The returned structure is a tree (providing that **ext_nodes** is defined using a complete tree structure – includes the optional **child_ptr** and **sibling_ptr** pointers).
- The root pointer is always the first sequential node (e.g., **ext_nodes[0]**).
- The node pointed to by **ext_node_ptr** is always the sequential starting node (e.g., **ext_nodes[ext_node_ptr]**) where traversal should continue. The external application is responsible for resetting **ext_node_ptr** to a node in the subtree from which traversal should continue in subsequent calls.
- The nodes in **ext_nodes** are returned sequentially using preorder. This tree has the characteristic that child pointers and sibling pointers are not required to traverse the tree. Preorder traversal is reduced to simple sequential access of nodes (e.g., **ext_nodes[n]**).

ext_nodes[n+1], ext_nodes[n+2], etc.).

- For each node in **ext_nodes**, the complete lineage of nodes back to the root node also exists in **ext_nodes**. This lineage can be accessed using **ext_node[n].parent_ptr**.
- While **Partial_Preorder_Traverse_Tree** is generally defined as an external interface, this type of interface can be used internally on the complete tree to defragment and garbage collect a highly fragmented tree structure as the returned structure has no unused sequential nodes. This defragmentation process requires only a single pass of the fragmented tree and the resulting tree is ordered using preorder.

The CONTROLLER process uses a tree traversal continuation mechanism for partial traversal of the A-Series process family tree when it makes calls on the A-Series MCP using GETSTATUS Type 6 Subtype 13 calls to retrieve process family information. The CONTROLLER process requests a subset of the process family information (a partial general tree) and consumes an even smaller subset of the retrieved information. For subsequent process family display requests, the CONTROLLER process sets up a specific process lineage (the last one displayed) and a pointer to that information as part of the next call to the GETSTATUS Type 6 Subtype 13 interface.

The following pseudocode represents the internal procedures that support two external interfaces defined to partially traverse dynamic general trees:

```
INTERFACE (Single_Step_Preorder_Traverse_Tree,
          Partial_Preorder_Traverse_Tree);
```

Procedure **Single_Step_Preorder_Traverse_Tree** returns a single node pointed to by **ext_node_ptr** that is the next preorder node in the tree. Subsequent traversal calls for continuation do not require update of **ext_node_ptr** by the calling program.

```
PROCEDURE Single_Step_Preorder_Traverse_Tree (ext_node_ptr, ext_nodes);
INTEGER ext_node_ptr BY REFERENCE;
ARRAY ext_nodes BY REFERENCE;
BEGIN
    Find_First_Node (ext_node_ptr, ext_nodes,
                    level, int_pointers);
    Insert_First_Lineage (ext_node_ptr, ext_ptr, ext_nodes, ext_pointers,
                        level, int_pointers);
END;
```

Two versions of procedure **Partial_Preorder_Traverse_Tree** are shown. The first version inserts nodes into the **ext_nodes** array until there is no more room in the array. The second version inserts a specific number of new nodes (**num_nodes**) of new nodes into the **ext_nodes** array. In both versions, **num_nodes** reports how many nodes were traversed. Subsequent traversal calls for continuation require update of **ext_node_ptr** by the calling program.

```

PROCEDURE Partial_Preorder_Traverse_Tree (ext_node_ptr, ext_nodes, num_nodes);
INTEGER ext_node_ptr, num_nodes BY REFERENCE;
ARRAY ext_nodes BY REFERENCE;
5 BEGIN
  INTEGER ext_ptr, level;
  BOOLEAN finished;
  ARRAY int_pointers, ext_pointers[0:maxlevels];

10  Find_First_Node (ext_node_ptr, ext_nodes,
                    level, int_pointers);
  Insert_First_Lineage (ext_node_ptr, ext_ptr, ext_nodes, ext_pointers,
                      level, int_pointers);

15  num_nodes := 1;
  finished := FALSE;
  WHILE NOT finished DO
    BEGIN
      Find_Next_Node (level, int_pointers, ext_pointers);
      IF level >= 0 THEN
20        BEGIN
          Insert_Next_Node (ext_ptr, ext_nodes, ext_pointers,
                          level, int_pointers);

          num_nodes := *+1;
          IF ext_ptr+1 = SIZE (ext_nodes) THEN
25            finished := TRUE; % no more nodes fit into external tree
        END
      ELSE
        finished := TRUE; % traversal of the tree finished
      END;
30 END;

PROCEDURE Partial_Preorder_Traverse_Tree (ext_node_ptr, ext_nodes, num_nodes);
INTEGER ext_node_ptr, num_nodes BY REFERENCE;
ARRAY ext_nodes BY REFERENCE;
35 BEGIN
  INTEGER ext_ptr, level, n;
  BOOLEAN finished;
  ARRAY int_pointers, ext_pointers[0:maxlevels];

40  Find_First_Node (ext_node_ptr, ext_nodes,
                    level, int_pointers);
  Insert_First_Lineage (ext_node_ptr, ext_ptr, ext_nodes, ext_pointers,
                      level, int_pointers);

  n := num_nodes;
45  num_nodes := 1;
  IF n < 1 THEN
    finished := FALSE
  ELSE
    finished := TRUE;
50  WHILE NOT finished DO
    BEGIN
      Find_Next_Node (level, int_pointers, ext_pointers);
      IF level >= 0 THEN
        BEGIN
55          Insert_Next_Node (ext_ptr, ext_nodes, ext_pointers,
                          level, int_pointers);

          n := *-1;
          num_nodes := *+1;
          IF n < 1 THEN
60            finished := TRUE; % returned <num_nodes> nodes
        END
      ELSE
        finished := TRUE; % traversal of the tree finished
      END;
65 END;
END;

```

The interface procedures **Single_Step_Preorder_Traverse_Tree** and **Partial_Preorder_Traverse_Tree** require four underlying procedures to accomplish the preorder traversal. Both of the interface procedures pass the lineage of the node from which to continue. This lineage is analogous to a recursion simulation stack that is used to eliminate recursive procedure calls. As such, the underlying internal procedures **Find_First_Node** and **Find_Next_Node** must use non-recursive techniques to traverse the internal general tree.

The procedure **Find_First_Node** processes the external array lineage to determine the node from which to continue. The interface must first determine the level of the node before constructing the recursion simulation stack (**int_pointers**). The main loop that builds the recursion simulation stack (**WHILE seeking DO**) compares nodes in the externally specified nodes (**ext_nodes**) versus the associated internal nodes (**nodes**). If nodes are determined to be matching (the node counters are equal), the level is incremented and the current pointer points to the associated child node. The counter values in the individual nodes are ordered (see procedure **Insert_Node**); in this example, an incrementing global node counter results in a decreasing counter value as one follows the sibling node chain. Thus, one can follow the sibling links to determine where the specified node fits. While following links (child links or sibling links), if the associated link is nil (not linked to a node), then the level is decremented and the current node pointer points to the next sibling node. The main loop is complete when the level is decremented to -1 (initial traversal call or traversal of the tree is complete) or continuation point is found.

```

PROCEDURE Find_First_Node (ext_node_ptr, ext_nodes, ext_pointers,
                           level, int_pointers);
  INTEGER ext_node_ptr, level BY REFERENCE;
  ARRAY ext_nodes, ext_pointers, int_pointers BY REFERENCE;
BEGIN
  ARRAY ancestor_nodes[0:maxlevels];
  INTEGER save_level, ptr;
  BOOLEAN seeking;

  % find depth of tree
  level := -1;
  ptr := ext_node_ptr;
  WHILE ptr <> nil DO
  BEGIN
    level := *+1;
    ptr := ext_node_array[ptr].parent_ptr;
  END;
  save_level := level;

  % retrieve specified lineage
  IF level >= 0 THEN
  BEGIN
    ancestor_nodes[level+1].counter := max_int;
    ptr := ext_node_ptr;
    WHILE ptr <> nil DO
    BEGIN
      ancestor_nodes[level] := ext_nodes[ptr];
      level := *-1;
      ptr := ext_node_array[ptr].parent_ptr;
    END;
    level := 0;
    int_pointers[level] := root_ptr;
  END;

  % establish continuation lineage (setup simulated recursion stack)
  seeking := TRUE;
  WHILE seeking DO
  BEGIN
    IF level < 0 THEN
    BEGIN % at end of tree or no start pointer - start at root
      seeking := FALSE;
      level := 0;
      int_pointers[level] := root_ptr;
    END;
  END;

```

```

END ELSE
IF int_pointers[level] = nil THEN
  BEGIN % no nodes at this level - drop back level and get next sibling
    level := *-1;
    int_pointers[level] := node[int_pointers[level]].sibling_ptr;
  END ELSE
  IF (node[int_pointers[level]].counter > ancestors[level].counter) THEN
    BEGIN % already visited this node - get next sibling
      int_pointers[level] := node[ptr].sibling_ptr;
    END ELSE
    IF (node[int_pointers[level]].counter = ancestors[level].counter) THEN
      BEGIN % node exists - increase level and get child
        level := *+1;
        int_pointers[level] := node[int_pointers[level-1]].child_ptr;
      END ELSE
      BEGIN % found first node at this level not yet visited
        seeking := FALSE;
      END;
    END;
  END;
END;

```

The procedure **Find_Next_Node** continues a preorder traversal of the tree using the recursion simulation stack (**int_pointers**). This procedure traverses the tree by following the child link. If the link is nil (child node does not exist), then the level is decremented and traversal continues by following the sibling link. If no sibling nodes exist at any level, the level is decremented and we follow that sibling link. The traversal ends when the level is decremented to -1.

```

PROCEDURE Find_Next_Node (level, int_pointers, ext_pointers);
INTEGER level BY REFERENCE;
ARRAY int_pointers, ext_pointers BY REFERENCE;
BEGIN
  % simulate recursion - traverse child subtree, then traverse sibling subtree
  level := *+1;
  int_pointers[level] := node[int_pointers[level-1]].child_ptr;
  WHILE (level >= 0) CAND
    (int_pointers[level] = nil) DO
    BEGIN
      ext_pointers[level] := nil;
      level := *-1;
      IF level => 0 THEN
        int_pointers[level] := node[int_pointers[level]].sibling_ptr;
      END;
    END;
  END;

```

Two procedures, **Insert_First_Lineage** and **Insert_Next_Node**, are defined to transfer internal tree nodes to the **ext_nodes** array. For each node transferred, the associated **parent_ptr** link must be "fixed up" as the external parent pointers differ in value from internal parent pointers.

Also if the optional node pointers (**child_ptr** and **sibling_ptr**) are defined in **ext_nodes**, these too must be "fixed up". However these pointers refer to nodes that have not yet been visited in traversal. A so-called inuse flag pointer value is defined that is essentially a place holder indicating future pointer resolution if the specific child or sibling is visited in this partial traversal. Thus, option node pointers may have 3 different types of values: a pointer to an existing node, a nil pointer (no node exists), and an inuse flag pointer (node exists internally but has not been inserted into the external partial tree).

The procedure **Insert_First_Lineage** follows the entire recursion simulation stack (**int_pointers**) and builds the associated external tree nodes. After creating the external nodes (**ext_nodes**), this procedure sets the external node pointer (**ext_node_ptr**) to point to the specific node from which to continue tree traversal.

```

5  PROCEDURE Insert_First_Lineage (ext_node_ptr, ext_ptr, ext_nodes, ext_pointers,
                                level, int_pointers);
    INTEGER ext_node_ptr, ext_ptr BY REFERENCE;
    INTEGER level BY VALUE;
    ARRAY ext_nodes, ext_pointers, int_pointers BY REFERENCE;
10  BEGIN
    ext_ptr := 0;
    ext_nodes[ext_ptr] := node[int_pointer[ext_ptr]];
    WHILE ext_ptr < level DO
15      BEGIN
        ext_ptr := *+1;
        ext_node[ext_ptr] := node[int_pointer[ext_ptr]];
        % fix links; set non-nil sibling pointers to "inuse" flag value
        IF ext_node[ext_ptr].sibling_ptr <> nil THEN
20          ext_node[ext_ptr].sibling_ptr := inuse;
          ext_node[ext_ptr-1].child_ptr := ext_ptr;
          ext_node[ext_ptr].parent_ptr := ext_ptr-1;
          ext_pointers[ext_ptr] := ext_ptr;
        END;
        % set non-nil child pointer to "inuse" flag value
25      IF ext_node[ext_ptr].child_ptr <> nil THEN
          ext_node[ext_ptr].child_ptr := inuse;
          ext_node_ptr := ext_ptr;
      END;
30  END;

```

The procedure **Insert_Next_Node** transfers a single internal node to **ext_nodes** and fixes the appropriate link values.

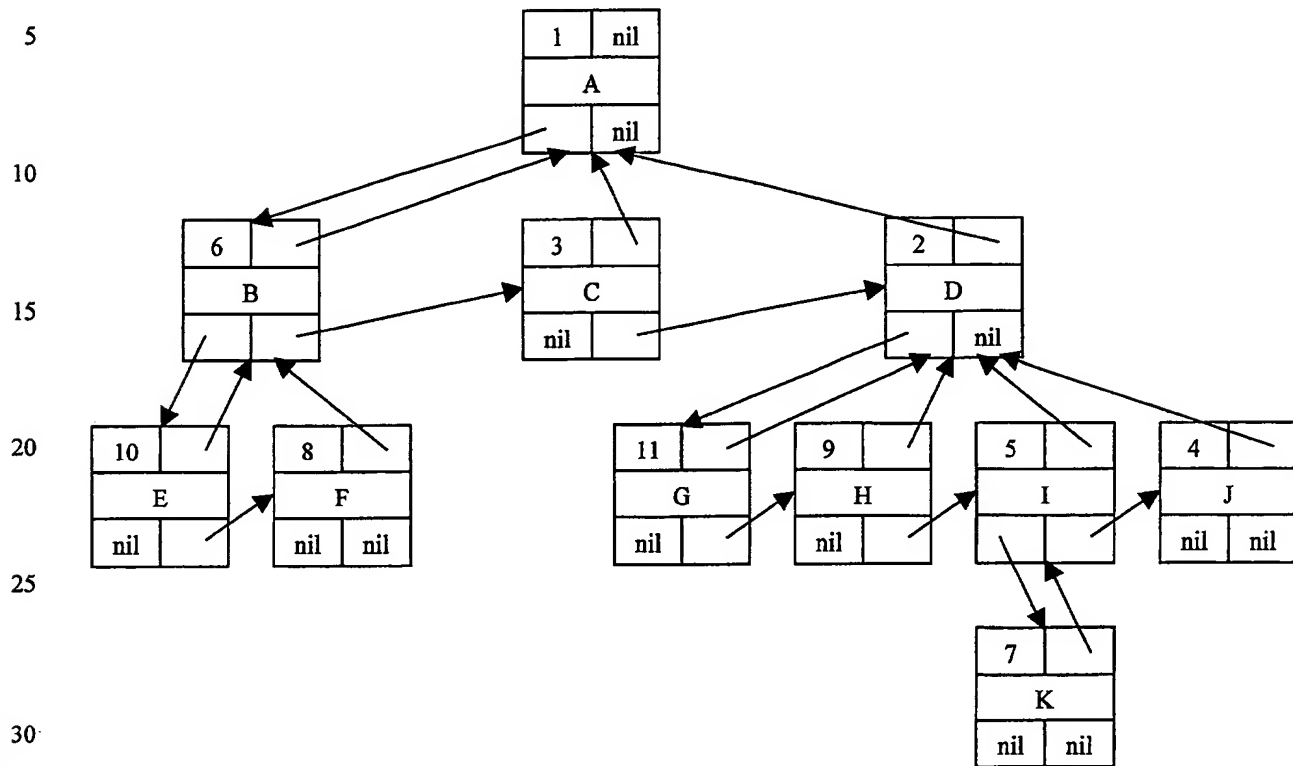
```

35  PROCEDURE Insert_Next_Node (ext_ptr, ext_nodes, ext_pointers, level, int_pointers);
    INTEGER ext_ptr BY REFERENCE;
    INTEGER level BY VALUE;
    ARRAY ext_nodes, ext_pointers, int_pointers BY REFERENCE;
    BEGIN
40      ext_ptr := *+1;
      ext_node[ext_ptr] := node[int_pointers[level]];
      % fix parent's link or previous sibling's link to point to this node
      IF ext_pointers[level] = nil THEN
        ext_node[ext_pointers[level-1]].child_ptr := ext_ptr
      ELSE
45        ext_node[ext_pointers[level]].sibling_ptr := ext_ptr;
        % set non-nil sibling pointer to "inuse" flag
        IF ext_node[ext_ptr].sibling_ptr <> nil THEN
          ext_node[ext_ptr].sibling_ptr := inuse;
        % set non-nil child pointer to "inuse" flag value
50        IF ext_node[ext_ptr].child_ptr <> nil THEN
          ext_node[ext_ptr].child_ptr := inuse;
        % set parent link
        ext_node[ext_ptr].parent_ptr := ext_pointers[level-1];
        ext_pointers[ext_ptr] := ext_ptr;
55      END;
    END;

```

This illustrates the use of external procedure **Partial_Preorder_Traverse_Tree** on the general tree in example 1 where 3 traversal nodes are requested for each call. Thus for 11 nodes, 4 calls are required.

The internal general tree structure and the associated traversal is:



Preorder Traversal: A B E F C D G H I K J

Procedure **Partial_Preorder_Traverse_Tree** call number 1:

5 Before the call

ext_node_ptr = nil

ext_nodes array = empty

10

After the call (sequential and graphic representation)

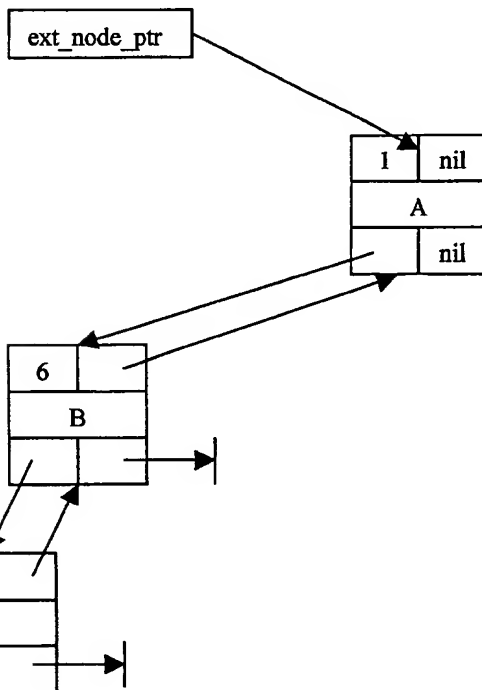
ext_node_ptr = 0

15

ext_nodes array

index	node_value	counter	parent_ptr	child_ptr	sibling_ptr
0	A	1	nil	1	nil
1	B	6	0	2	inuse
2	E	10	1	nil	inuse

20



25

30

35

40

45

Call number 1 Preorder Traversal: A B E

Procedure **Partial_Preorder_Traverse_Tree** call number 2:

5 Before the call

ext_node_ptr = 2 (set by the calling program)**ext_nodes** array

index	node_value	counter	parent_ptr	child_ptr	sibling_ptr
0	A	1	nil	1	nil
1	B	6	0	2	inuse
2	E	10	1	nil	inuse

10

After the call (sequential and graphic representation)

ext_node_ptr = 2

15

ext_nodes array

index	node_value	counter	parent_ptr	child_ptr	sibling_ptr
0	A	1	nil	1	nil
1	B	6	0	2	3
2	F	8	1	nil	nil
3	C	3	0	nil	4
4	D	2	0	inuse	nil

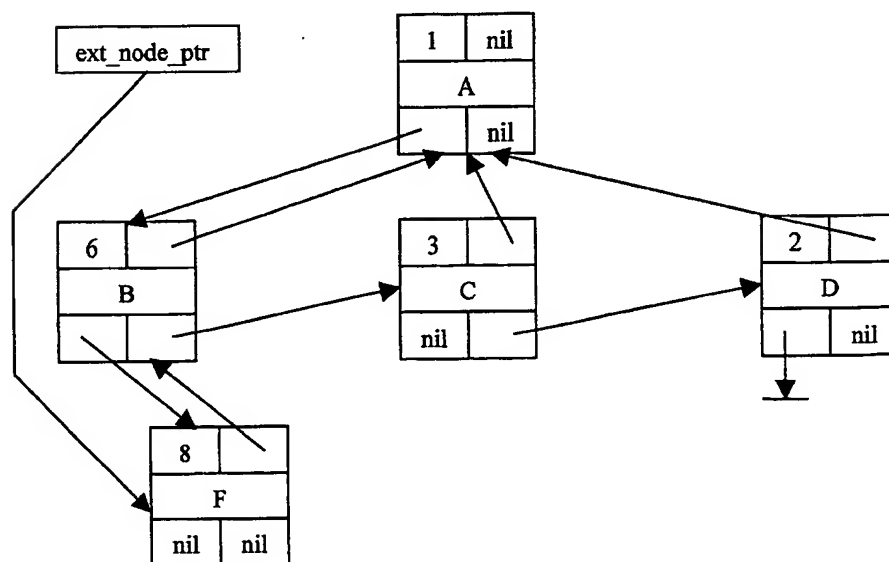
20

25

30

35

40



Call number 1 Preorder Traversal: A B E

Call number 2 Preorder Traversal: F C D

Procedure **Partial_Preorder_Traverse_Tree** call number 3:

5 Before the call

ext_node_ptr = 4 (set by the calling program)

ext_nodes array

index	node_value	counter	parent_ptr	child_ptr	sibling_ptr
0	A	1	nil	1	nil
1	B	6	0	2	3
2	F	8	1	nil	nil
3	C	3	0	nil	4
4	D	2	0	inuse	nil

10

After the call (sequential and graphic representation)

ext_node_ptr = 2

ext_nodes array

index	node_value	counter	parent_ptr	child_ptr	sibling_ptr
0	A	1	nil	1	nil
1	D	2	0	2	nil
2	G	11	1	nil	3
3	H	9	1	nil	4
4	I	5	1	inuse	inuse

15

20

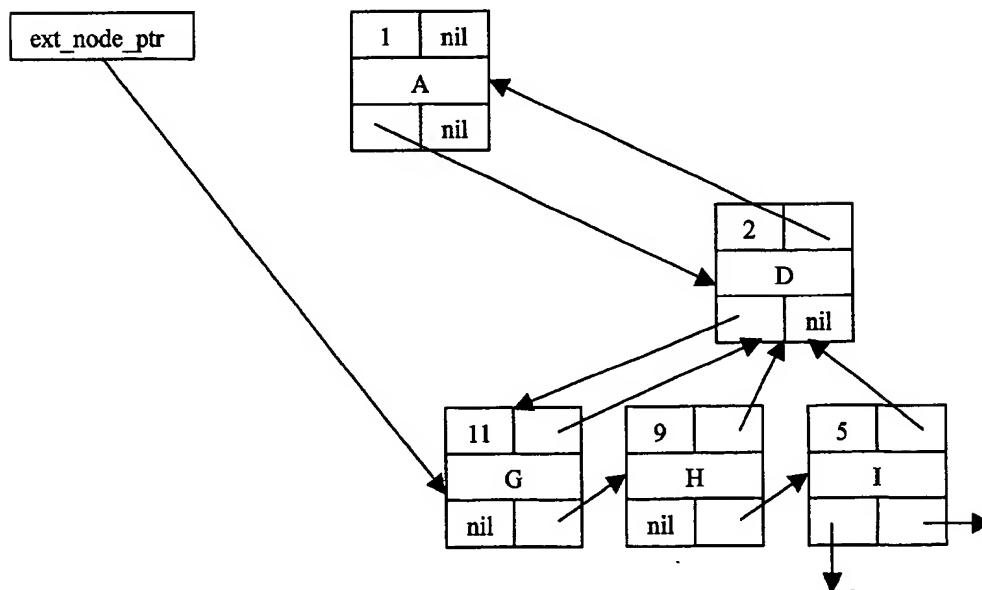
25

30

35

40

45



Call number 1 Preorder Traversal: A B E
 Call number 2 Preorder Traversal: F C D
 Call number 3 Preorder Traversal: G H I

Procedure **Partial_Preorder_Traverse_Tree** call number 4:

5 Before the call

ext_node_ptr = 4 (set by the calling program)

ext_nodes array

index	node_value	counter	parent_ptr	child_ptr	sibling_ptr
0	A	1	nil	1	nil
1	D	2	0	2	nil
2	G	11	1	nil	3
3	H	9	1	nil	4
4	I	5	1	inuse	inuse

10

After the call (sequential and graphic representation)

ext_node_ptr = 3

ext_nodes array

index	node_value	counter	parent_ptr	child_ptr	sibling_ptr
0	A	1	nil	1	nil
1	D	2	0	2	nil
2	I	5	1	3	4
3	K	7	2	nil	nil
4	J	4	1	nil	nil

15

20

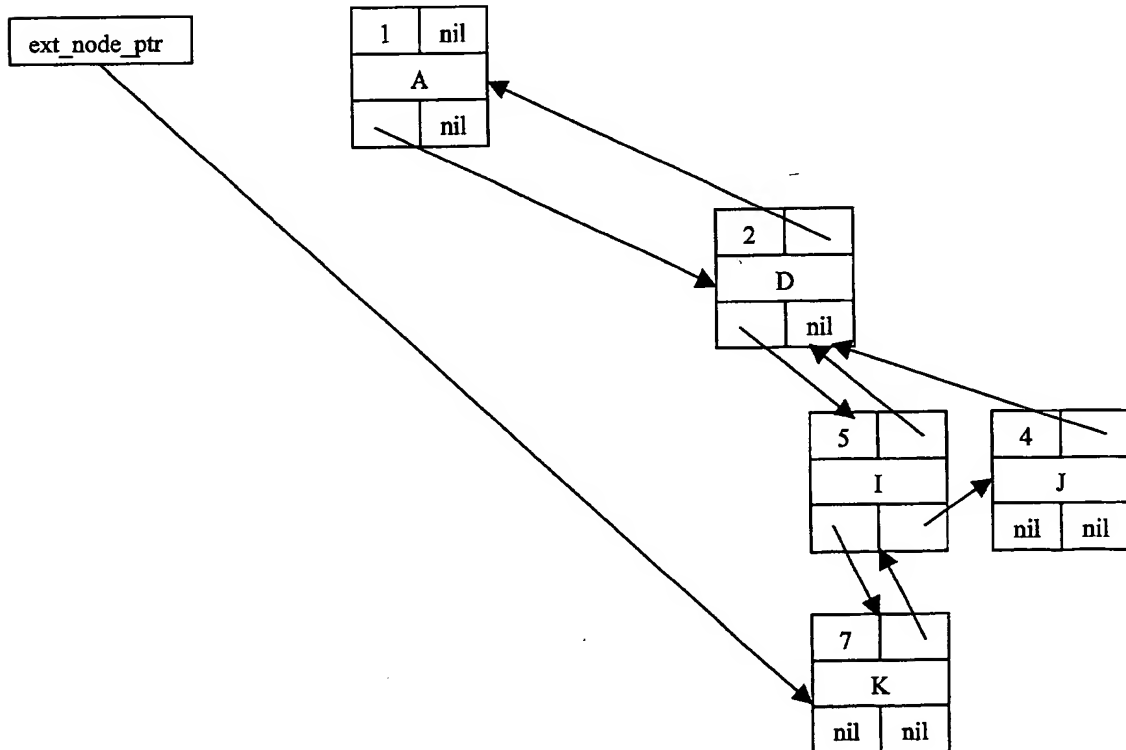
25

30

35

40

45



5

Call number 1 Preorder Traversal: A B E
Call number 2 Preorder Traversal: F C D
Call number 3 Preorder Traversal: G H I
Call number 4 Preorder Traversal: K J

Concatenated Preorder Traversal: A B E F C D G H I K J

5 **Residences of Primary Contributor(s):**

Philip M. Hoffman	USA	301 Paper Mill Road	Oreland / Montgomery	PA / USA / 19075
(Name)	(Citizenship)	(Street Address)	(City/County)	(State/Country/Zip Postal Code)
Andrew David Milligan	United Kingdom	10 Denison Court Wavendon Gate	Milton Keynes	United Kingdom / MK7 7JF
(Name)	(Citizenship)	(Street Address)	(City/County)	(State/Country/Zip Postal Code)
Robert K. Liermann	USA	1317 Kerwood Lane	Downingtown / Chester	PA / USA / 19335
(Name)	(Citizenship)	(Street Address)	(City/County)	(State/Country/Zip Postal Code)
Clark C. Kogen	USA	1290 South Creek Road	West Chester / Chester	PA / USA / 19382
(Name)	(Citizenship)	(Street Address)	(City/County)	(State/Country/Zip Postal Code)

Abstract Data Types

Specifications, Implementations, and Applications

Nell Dale
The University of Texas at Austin
Henry M. Walker
Grinnell College

D. C. Heath and Company
Lexington, Massachusetts Toronto

Address editorial correspondence to

D.C. Heath and Company

125 Spring Street

Lexington, MA 02173

Acquisitions: Walter Cunningham

Development: Rebecca Johnson, Karen H. Jolie

Editorial Production: Heather Garrison, Anne Starr

Design: Henry Rachlin

Production Coordination: Charles Dutton

Copyright © 1996 by D.C. Heath and Company.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage or retrieval system, without permission in writing from the publisher.

Published simultaneously in Canada.

Printed in the United States of America.

International Standard Book Number: 0-669-35444-9

Library of Congress Catalog Number: 95-68945

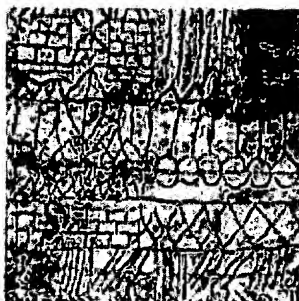
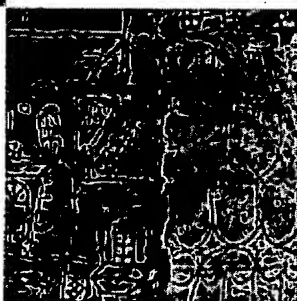
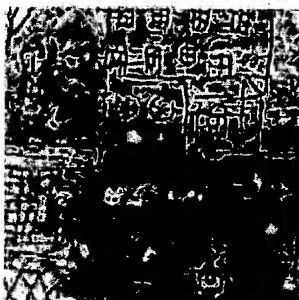
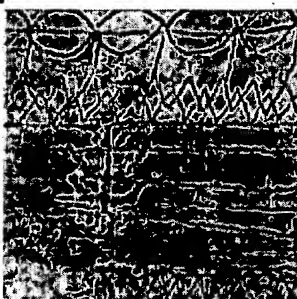
10 9 8 7 6 5 4 3 2 1

7—

Binary Trees

In the first part of this book, we encountered a "chicken and egg" problem—that is, we needed a way in which to implement lists of items before we actually got around to defining a list as an abstract data type. Also, we implemented the ADT List in five ways. Four of the primitive implementations were linear; the fifth was a binary search tree implementation. We have not, however, relied upon your understanding of binary search trees thus far.

This chapter begins by defining the notion of general trees and introducing some terminology. Attention then focuses upon a particular type of tree, called a binary tree. The formal specifications for the ADT BinTree organize data in a hierarchical structure, but there need not be any special ordering relationship among the data. Implementations for binary trees include traditional approaches with pointers or arrays as well as threaded trees. Decision trees, an application of general binary trees, provide a useful way of viewing program execution and are a powerful tool for determining the best possible efficiency for solving some types of problems. The chapter closes by considering heaps as a special type of binary tree, giving formal specifications, describing a particularly efficient array implementation, and outlining the use of heaps to implement priority queues.



General Trees

The ADT Binary Tree is an abstract data type that models the mathematical object *tree* or *rooted tree*. A tree is defined recursively, as follows:

1. A set of zero objects is a tree, called the *empty tree* or the *null tree*.
2. If T_1, T_2, \dots, T_n are n trees for $n \geq 0$ and R is an object, called a *node*, then the set T containing R and the trees T_1, T_2, \dots, T_n is a tree. Within T , R is called the *root* of T and T_1, T_2, \dots, T_n are called *subtrees*.

When we visualize a tree, we use a box to represent the external name of the tree and circles, boxes, or text to represent the nodes. Lines are drawn from a root to the roots of each of its subtrees. The representation of a tree is shown in Figure 7.1. For reference, we also have labeled each node in these trees.

The tree in Figure 7.1 (a) is the empty tree; there are no nodes. The tree in (b) has only one node, the root. The tree in (c) has 16 nodes. The root node has four subtrees. The roots of these subtrees are called the *children* of the root. Because our definition is recursive, each of the roots of these subtrees has subtrees. There are 16 nodes in the tree, so there are 15 nonempty subtrees. The nodes with no subtrees are called *terminal nodes* or, more commonly, *leaves*. There are 10 leaves

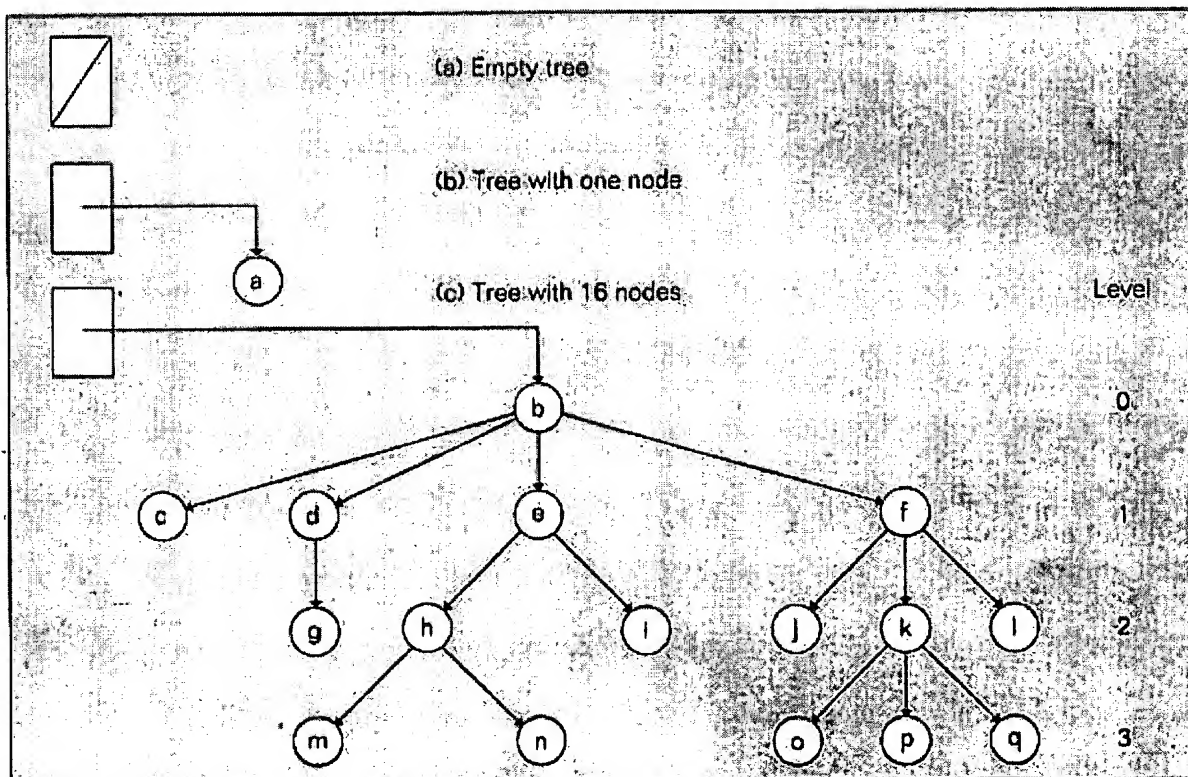


Figure 7.1
Trees with Nodes Labeled with Letters

in the tree in (c). The *degree of a node* is the number of subtrees that it has. Thus, the degrees of the nodes in Figure 7.1 range from zero to four. By definition, the degree of each leaf node is zero. The *degree of a tree* is the maximum degree of a node in the tree. As the tree in (a) has no nodes, there is no maximum degree of a node, and the degree of the tree is not defined. The tree in (b) has degree 0, and the tree in (c) has degree four.

Because family relationships can be modeled as trees, we often call the root of a tree (or subtree) the *parent*, and the roots of the subtrees the *children*. Consequently, the children of a node are called *siblings*.

A great deal of processing takes advantage of the relationship between a parent and its children, and we commonly say a *directed edge* (or, simply, an *edge*) extends from a parent to its children. Thus, edges connect a root with the roots of each subtree. For example, in Figure 7.1 (c), an edge extends from the root b to each of the nodes c, d, e, and f. Similarly, edges extend from e to i and from k to q. An *undirected edge* extends in both directions between a parent and a child. Thus, undirected edges also would extend from i to e and from q to k.

A *directed path* (or, simply, *path*) is a sequence of directed edges e_1, e_2, \dots, e_n where the node at the end of one edge serves as the beginning of the next edge. An *undirected path* is a similar sequence of undirected edges.

For example, in Figure 7.1 (c) one path containing three edges begins at the root and extends through nodes f, k, and q. Similarly, the path beginning with node h and containing nodes e, b, and d is an undirected path. In this chapter and the next two chapters, when we say *edge* we mean a *directed edge* from a parent to its child. (In Chapter 10, we again introduce the concept of an undirected edge.) Following the analogy of family hierarchies, if a path exists from one node to another, it is common to state that the first node is an *ancestor* of the second, and the second is a *descendant* of the first.

The *length* of a path is the number of edges it contains (which is one less than the number of nodes on the path). The *depth* or *level* of a node is the length of the directed path from the root to that node. The *height* of a tree is the length of the path from the root to a node on the lowest level. Thus, the level of the root of a tree is zero, and the level of each child of the root is one. Equivalently, the height of a tree is the largest level number of any node in the tree.¹

There are three common ways to systematically order (or list) the nodes in a tree: *preorder*, *inorder*, and *postorder*. For each of these orderings, the empty tree gives rise to the empty list, and the tree with one node yields the list with one node. For trees with more than one node, the following statements are true.

- The preorder list contains the root, followed by the preorder list of the nodes of the subtrees of the root from left to right.
- The inorder list contains the inorder list of the left-most subtree, the root, and the inorder list of each of the other subtrees from left to right.

¹ Some texts designate the level of the root to be one. This changes the relation of the level number in various definitions but does not change the definitions themselves.

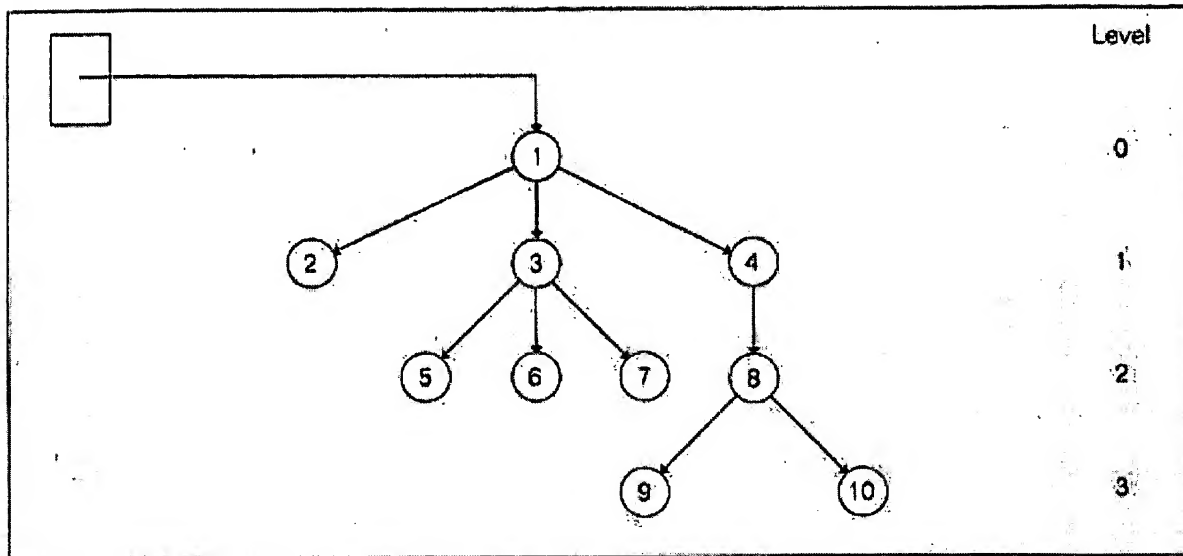


Figure 7.2
Tree with Nodes Labeled with Numbers

- The postorder list contains the postorder list of the subtrees of the root from left to right, followed by the root.

Figure 7.2 shows a tree whose nodes are labeled with numbers rather than letters. Following is a list of terms for review, using a concrete example from the tree in Figure 7.2.

Subtrees The nodes labeled 2, 3, and 4 are the roots of the subtrees (children) of the node labeled 1. The nodes labeled 5, 6, and 7 are the roots of the subtrees (children) of the node labeled 3; the node labeled 8 is the root of the subtree (child) of the node labeled 4. The nodes labeled 9 and 10 are the roots of the subtrees (children) of the node labeled 8.

Leaves The nodes labeled 2, 5, 6, 7, 9, and 10 are terminal nodes or leaf nodes.

Degree The nodes labeled 1 and 3 have degree 3. The node labeled 8 has degree 2. The node labeled 4 has degree 1. All the leaf nodes have degree 0. The degree of the tree is 3, because the maximum degree of any node is 3.

Levels The level numbers appear on the right of the tree. The level of the root is 0, the level of the nodes labeled 2, 3, and 4 is 1, the level of the nodes labeled 5, 6, 7, and 8 is 2, and the level of the nodes labeled 9 and 10 is 3.

Family relationships The node labeled 1 is the parent of the nodes labeled 2, 3, and 4. The node labeled 3 is the parent of the nodes labeled 5, 6, and 7. The node labeled 4 is the parent of the node labeled 8, which, in turn, is the parent of the nodes labeled 9 and 10. The nodes labeled 2, 3, and 4 are siblings. The nodes labeled 5, 6, and 7 are siblings. The nodes labeled 9 and 10 are siblings. Note that the node labeled 8 is not the sibling of the nodes labeled 5, 6, and 7.

Paths and path lengths Paths exist from all parents to children. A unique path exists from the root to each leaf node; these are shown below. The length of each path is also shown. Because any subpath is a path, all of the paths are represented.

1 → 2 Length: 1

1 → 3 → 5 Length: 2

1 → 3 → 6 Length: 2

1 → 3 → 7 Length: 2

1 → 4 → 8 → 9 Length: 3

1 → 4 → 8 → 10 Length: 3

Height and depth The height of the tree is 3, the maximum level. The depth of the nodes labeled 2, 3, and 4 is 1. The depth of the nodes labeled 5, 6, 7, and 8 is 2. The depth of the nodes labeled 9 and 10 is 3, the same as the height of the tree. The depth of the nodes on the lowest level is always the same as the height of the tree.

Orderings The preorder, inorder, and postorder orderings of the nodes are given below.

1 → 2 → 3 → 5 → 6 → 7 → 4 → 8 → 9 → 10 (preorder)

2 → 1 → 5 → 3 → 6 → 7 → 9 → 8 → 10 → 4 (inorder)

2 → 5 → 6 → 7 → 3 → 9 → 10 → 8 → 4 → 1 (postorder)

Binary trees are trees where the maximum degree of any node is two. Any general tree can be represented as a binary tree using the following algorithm.

1. Insert an arrow from each node to its right sibling (if one exists).
2. Remove arrows from each node to all but the left-most child.

We can apply this algorithm to the trees in Figure 7.1. The tree in (a) is empty, so there are no nodes, no siblings, no children, and, therefore, nothing to do: the binary representation of an empty general tree is an empty tree. The tree in (b) has one node, but no siblings and no children, so there is nothing to do. The binary representation of a one-node general tree is a one-node binary tree. The tree in (c) with these changes is shown on the next page in Figure 7.3.

While this, in fact, is a binary tree, it may not look like a usual tree structure. Think of the nodes in the figure as being connected by string and pick up the tree by the root. The result is Figure 7.4, which may look more familiar.

In comparing the trees in figures 7.1 and 7.4, both have 16 nodes—so far so good. Leaf nodes in the binary representation (Figure 7.4) should be those leaf nodes in the original tree that had no siblings to the right. Of the 10 leaf nodes in the original, 5 had no right siblings; there are 5 leaf nodes in the binary representation. Each left node in the binary representation was a left-most child; there

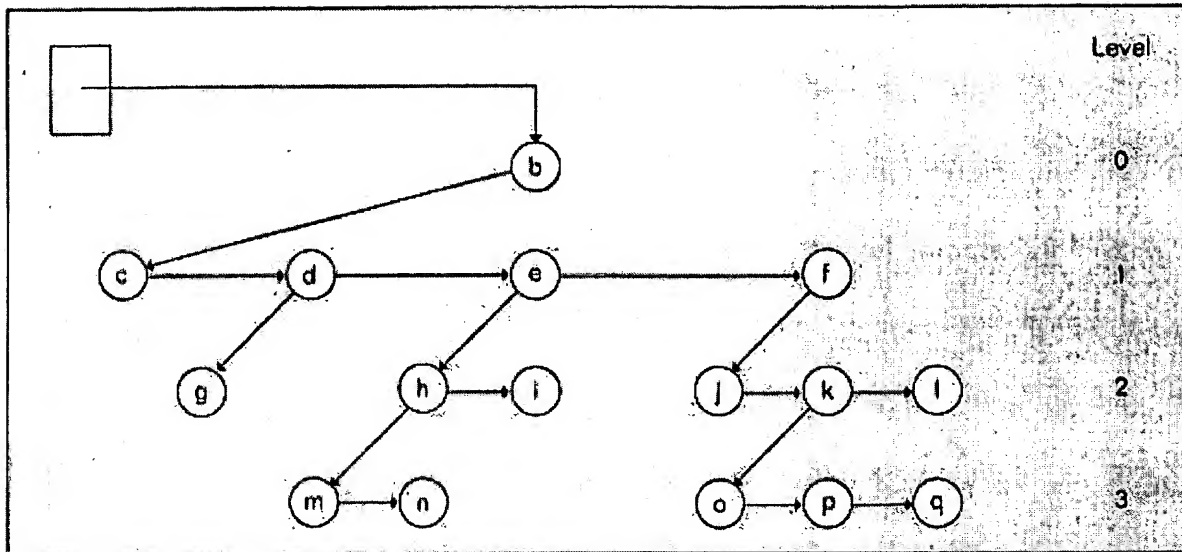


Figure 7.3
General Tree to Binary Tree

were 6 left-most children in the original and there are 6 left nodes in the binary representation. As a second example, we apply the algorithm to the labeled tree in Figure 7.2, resulting in Figure 7.5.

Notice that the transformation from Figure 7.2 to Figure 7.5 is reversible. That is, given a binary representation of a general tree, we can re-create the general tree. A left node is the left-most child of its parent. A right node is a sibling of its parent.

Binary trees have certain interesting properties. Because the maximum degree of any node is 2, we can determine the maximum number of nodes at any level: 2^k where k is the level number. A *full binary tree* is a binary tree where all of the leaves are on the same level and that level has the maximum number of leaves. A full binary tree has $2^{k+1} - 1$ nodes. A count of nodes in a binary tree is illustrated in Figure 7.6.

A *complete binary tree* is either a full tree or a tree that is full through the maximum level minus one, and the nodes on the last level are as far to the left as possible. The tree in Figure 7.6 is full through the third level, and is complete.

Binary Trees

As in the case of arrays and sequences and unsorted and sorted lists, there are two types of binary trees: one where there is only a store operation and one where there is also an insert operation. Here, we consider binary trees with only the Store operation. Discussions later in this chapter and in Chapter 8 cover binary trees with an Insert operation (heaps and binary search trees).

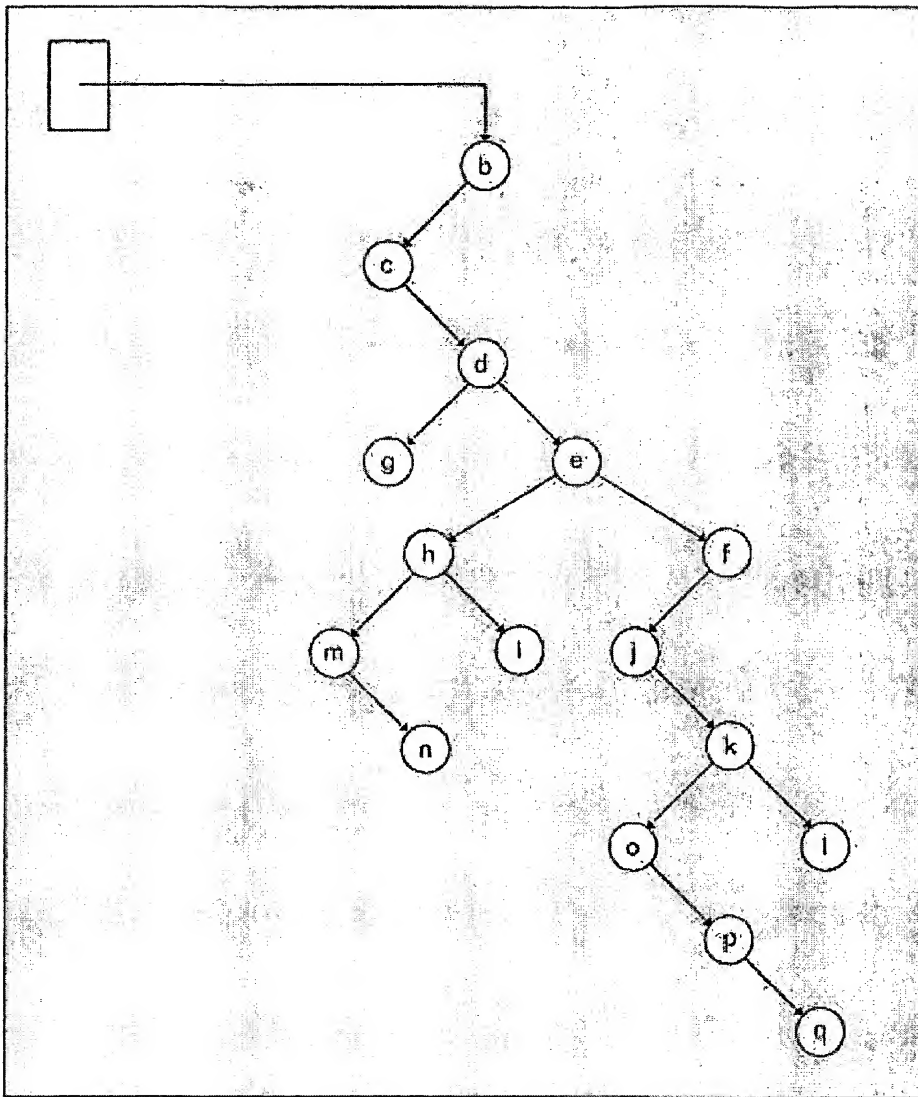


Figure 7.4
The Binary Tree of Figure 7.3 in a More Familiar Format

Specification

structure BinTree (of ItemType)

interface Create → BinTree

Make(BinTree, ItemType, BinTree) → BinTree

LeftTree(BinTree) → BinTree

RightTree(BinTree) → BinTree

Item(BinTree) → ItemType

IsEmpty(BinTree) → Boolean

end

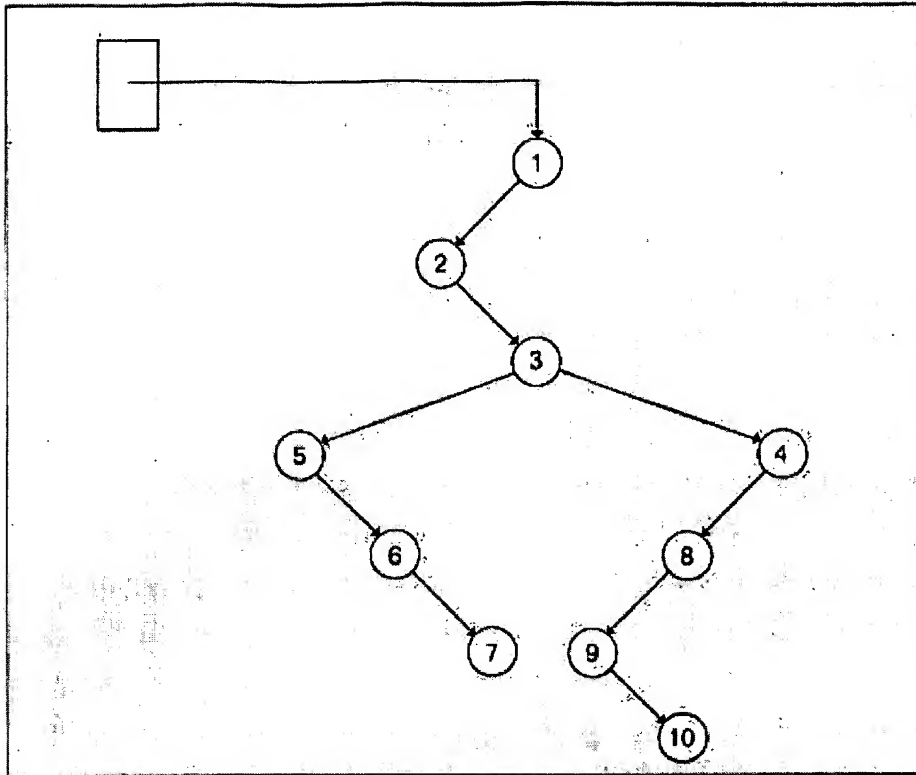


Figure 7.5
Tree from Figure 7.2 as a Binary Tree

axioms for all BT1, BT2 in BinTree, i1 in ItemType, let

LeftTree(Create) = Error

LeftTree(Make(BT1, i1, BT2)) = BT1

RightTree(Create) = Error

RightTree(Make(BT1, i1, BT2)) = BT2

Item(Create) = Error

Item(Make(BT1, i1, BT2)) = i1

IsEmpty(Create) = True

IsEmpty(Make(BT1, i1, BT2)) = False

end

end BinTree

In later discussions, we need to identify the elements within a tree in a standard way. Assuming the values in the nodes have an intrinsic ordering of some type, this can be done by placing the tree elements in a nonindexed, sorted list, as specified in Chapter 6. (For example, if our tree contains strings, then the strings can be inserted into a sorted list in alphabetical order. Similarly, a natural ordering is available if the nodes contain numbers.) This gives rise to the following defi-

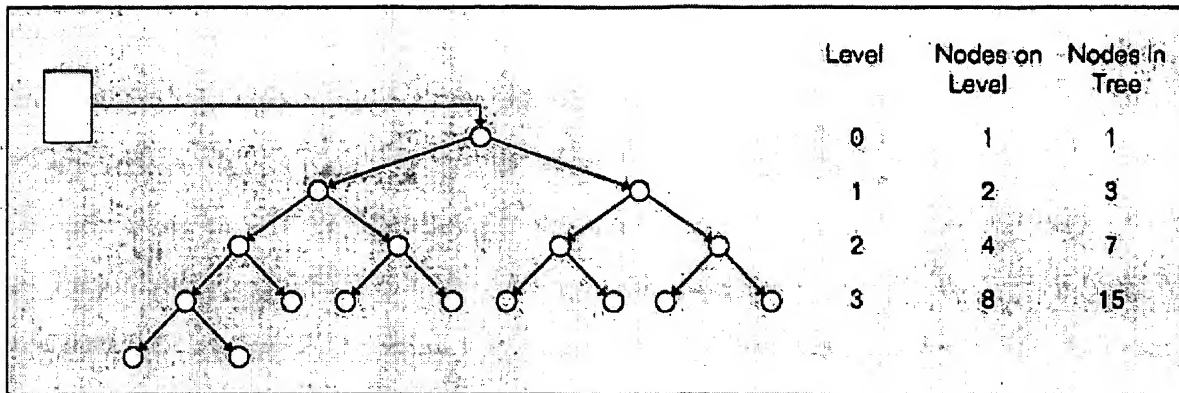


Figure 7.6
Node Count in a Binary Tree

nitions, which use the Create and Insert operations of the SortedList ADT. We also need the definition of the operation Merge, which combines two sorted lists to give a new sorted list.

Merge (SortedList, SortedList) \rightarrow SortedList
 IdentifyTreeElements (BinTree) \rightarrow SortedList

Merge (Create, L2) = L2
 Merge (Make (L1, i1), L2) = Merge (L1, Insert(L2, i1))

IdentifyTreeElements (Create) = Create
 IdentifyTreeElements (Make(BT1, i1, BT2))
 = Insert(Merge(IdentifyTreeElements(BT1),
 IdentifyTreeElements (BT2)), i1)

This last axiom states that we can obtain the sorted list of tree elements for the tree Make (BT1, i1, BT2) by merging the lists obtained from the subtrees and then inserting element i1. (Recall that the Insert operation on sorted lists adds an element to a list in an appropriate location to maintain the ordering of the new list.)

Given two binary trees, the IdentifyTreeElements operation provides a relatively simple way to determine if the elements in the two trees are the same (although the organization of the elements within the trees may be quite different). We simply compare the elements in the two sorted lists. This gives rise to the operation SameElements. As an auxiliary function, we define the operation EqualLists, which determines if two sorted lists are the same.

EqualLists (SortedList, SortedList) \rightarrow Boolean
 SameElements (BinTree, BinTree) \rightarrow Boolean

EqualLists (Create, Create) = True
 EqualLists (Make(L1, i1), Create) = False


```

EqualLists (Create, Make(L2, i2)) = False
EqualLists, (Make(L1, i1), Make(L2, i2)) =
  IF i1 = i2
    THEN EqualLists (L1, L2)
    ELSE False
  END IF

```

```

SameElements (BT1, BT2) =
  EqualLists(IdentifyTreeElements(BT1), IdentifyTreeElements (BT1))

```

The operation SameElements provides a compact way of comparing the elements within two binary trees. The definition of SameElements requires the use of the SortedList ADT and several supplementary operations, such as IdentifyTreeElements and EqualLists. These definitions seem somewhat complex, so you may wonder why we did not define SameElements directly in terms of binary tree operations, avoiding the use of sorted lists completely. While such an approach is possible, it is even more complicated than the approach shown here. For example, binary trees can have any shape, and the axioms do not dictate where elements might be. Also, duplicate data values may be present. The exercises at the end of this chapter pursue this direct definition of SameElements further.

If we want to list all of the elements in the tree systematically, then we need to define an iterator operation. In the following, we give the axiomatic specifications for a preorder traversal utilizing the Queue ADT (of ItemType) augmented with the operation Concat, which concatenates two Queues. The end of chapter exercises ask you to write the specifications for the inorder and postorder traversals.

```

Concat(Queue, Queue)      → Queue
PreOrder(BinTree)         → Queue

```

```

Concat (Q, Create) = Q
Concat (Q, Enqueue(P, i1)) = Enqueue (Concat (Q, P), i1)

```

```

PreOrder(Create) = Create
PreOrder(Make(BT1, i1, BT2)) =
  Concat(Concat(Enqueue(Create, i1), PreOrder(BT1)), PreOrder(BT2))

```

Note carefully the difference between the list returned from IdentifyTreeElements and the queue returned from PreOrder. The list from IdentifyTreeElements is a listing of the values in the nodes in the tree ordered by the intrinsic ordering of the values themselves; the queue from PreOrder lists the values in the nodes in the order in which the nodes are visited in a preorder traversal. (For a binary search tree, which we discuss in Chapter 8, the list from IdentifyTreeElements and the queue returned from an inorder traversal are the same.)

Implementation

The interface section of the ADT Binary Tree module is shown below. The data structure is hidden within the implementation section of the module. The opera-

tions are all implemented as functions. BinTree can either be an opaque type, or the definition of Bin Tree can be filled in at a later time.

Interface Section ADTBinary Tree;

(* To access ItemType. *)
USES < data defining module >

TYPE

BinTree; (* Opaque type or filled in later. *)

FUNCTION Create: BinTree

(* Post: Returns a Tree. *)

FUNCTION Make(Left: BinTree; Item: ItemType; Right: BinTree):
BinTree

(* Pre: Left and Right have been initialized. *)

(* Post: Returns a Tree with Left as Left(Make), Right as Right
(Make) *)

(* and Item as Item(Make). *)

FUNCTION LeftTree(Tree : BinTree): BinTree

(* Pre: NOT IsEmpty(Tree). *)

(* Post: Returns Left subtree of Tree. *)

FUNCTION RightTree(Tree : BinTree): BinTree

(* Pre: NOT IsEmpty(Tree). *)

(* Post: Returns Right subtree of Tree. *)

FUNCTION Item(Tree : BinTree): ItemType

(* Post: Returns Item field of Tree. *)

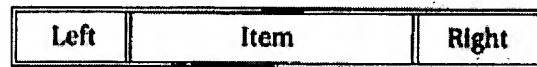
(* Pre: NOT IsEmpty(Tree). *)

FUNCTION IsEmpty(Tree: BinTree): Boolean

```
(* Pre:      Tree has been initialized.                                *)
```

(* Post: Returns True if Tree is empty; False otherwise. *)

Because most modern programming languages have pointer variables, it may seem natural to discuss immediately the implementation of a binary tree as a dynamic structure using referenced variables. However, at this stage we resist this approach, because such details are always machine- or language-dependent. A more universal approach uses node operations instead, at an intermediate level of abstraction. Our node of type `TreeNodeType` contains three fields: `Left`, `Item`, and `Right`.



The following node manipulation functions are available. We define the algorithms later.

FUNCTION GetNode: TreeNodeType

(* Post: Returns a Node. *)

FUNCTION GetLeft(Node: TreeNodeType): TreeNodeType

(* Post: Returns the Left field of Node. *)

FUNCTION GetRight(Node: TreeNodeType): TreeNodeType

(* Post: Returns the Right field of Node. *)

FUNCTION GetItem(Node: TreeNodeType) : ItemType

(* Post: Returns the Item field of Node. *)

PROCEDURE SetLeft(VAR Node1: TreeNodeType; Node2: TreeNodeType)

(* Post: Left(Node1) \leftarrow Node2. *)

PROCEDURE SetRight(VAR Node1: TreeNodeType; Node2: TreeNodeType)

(* Post: Right(Node1) \leftarrow Node2. *)

PROCEDURE SetItem(VAR Node: TreeNodeType; Item: ItemType)

(* Post: Item(Node) \leftarrow Item. *)

FUNCTION NULL(Node: TreeNodeType): Boolean

(* Post: Returns True if Node is empty; False otherwise. *)

PROCEDURE FreeNode(Node: TreeNodeType)

(* Post: Returns an unneeded Node. *)

We use these operations to write the algorithms for the binary tree operations.

Create: BinTree

RETURN NULL

Make(Left: TreeNodeType, Item: ItemType, Right: TreeNodeType): BinTree;

```
TempNode ← GetNode
SetLeft(TempNode, Left)
SetRight(TempNode, Right)
SetItem(TempNode, Item)
RETURN TempNode
```

LeftTree, RightTree, Item, and IsEmpty are directly coded as the node primitives GetLeft, GetRight, GetItem, and NULL.

If we use pointer variables, GetNode is implemented as

```
New(Ptr)
RETURN Ptr
```

and NULL implemented as

```
RETURN (Node = NIL)
```

However, we may also implement the tree structure using an array of records rather than pointer variables. In this approach, a node is stored as an entry within a large, predefined array, and an array subscript designates individual nodes. This approach parallels the work done by the operating system in allocating space dynamically for objects designated by pointer variables. Here, GetNode identifies a new location not currently designated within the array, while FreeNode designates that an array entry is no longer needed. The pool of nodes used in building the tree is described by the following declarations.

```
CONST
    NULL = 0;

TYPE

    TreeNode= 0..MaxNodes;
    BinTree = TreeNode;

    TreeNodeType = RECORD

        Left, Right : BinTree;
        Item        : ItemType

    END;

    NodesType = ARRAY [1..MaxNodes] OF
    TreeNodeType;

VAR

    Nodes        : NodesType;
    Avail        : BinTree;
```

Nodes is a pool of record variables of type TreeNode, and we can think of an array index as the address of a node within the array that represents storage. For example, Nodes[Ptr].Left and Nodes[Ptr].Right are logically equivalent to Ptr.Left and Ptr.Right in a pointer implementation.